6.1 An Introduction to Number Theory

We've spent the first five chapters of this course studying programming in Python. We've been mainly focused on how we represent data and designing functions to operate on this data. Up to this point, the *ideas* behind the functions that we've written have been relatively straight-forward, and the challenge has been in implementing these ideas correctly using various programming techniques. Over the next two chapters, we are going to study algorithms where the ideas themselves will be more complex. It won't be "obvious" how or why these algorithms work, and so to convince ourselves that these algorithms are correct, we'll study the formal mathematics behind them.

Our first large example of this is one that will take us the next two chapters to develop: the RSA cryptosystem, consisting of a pair of algorithms that are central to modern Internet security. If you haven't heard about RSA, cryptosystems, or ever thought about security, don't worry, we'll be building all of these concepts from the ground up over the course of this chapter and the next. What will set this apart from the kind of work we've done so far is that to understand what these algorithms do and why they work, we'll need to step away from code and into the realm of *number theory*, the branch of mathematics concerned with properties of integers.

We'll start our journey here with a few key definitions, some of which you've seen before defined formally in this course, and others that you might have heard about before, but not seen a formal definition.

Divisibility, primality, and the greatest common divisor

Here are our first two definitions; these are repeated from 3.9 Working with Definitions.

Definition. Let $n, d \in \mathbb{Z}$. We say that d **divides** n when there exists a $k \in \mathbb{Z}$ such that $n = dk$. We use the notation $d \mid n$ to represent the statement " d divides n".

The following phrases are synonymous with "d divides n":

- n **is divisible by** d
- d is a **factor** of n
- n is a **multiple** of d

Definition. Let $p \in \mathbb{Z}$. We say p is **prime** when it is greater than 1 and the only natural numbers that divide it are 1 and itself.

The next few definitions introduce and expand on the notion of common divisors between two numbers.

Definition. Let $x, y, d \in \mathbb{Z}$. We say that d is a **common divisor** of x and y when d divides x and d divides y.

We say that d is the **greatest common divisor** of x and y when it the largest number that is a common divisor of x and y, or 0 when x and y are both 0.[1] We can define the function

[1] According to this definition, what is $\gcd(0, n)$ when $n > 0$?

$\gcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ as the function which takes numbers x and y, and returns their greatest common divisor.

You might wonder whether this definition makes sense in all cases: is it possible for two numbers to have no divisors in common? One of the statements we will prove later in this chapter is that 1 divides every natural number. So at the very least, 1 is a common divisor between any two natural numbers. There is a special case, when 1 is the *only* positive divisor between two numbers.

Definition. Let $m, n \in \mathbb{Z}$. We say that m and n are **coprime** when $\gcd(m, n) = 1$.

Quotients and remainders

The next definitions are introduced through a fundamental theorem in number theory, which extends the relationship of divisibility to that of remainders.

Theorem. (Quotient-Remainder Theorem) For all $n \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$, there exist $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $0 \leq r < d$. Moreover, these q and r are *unique* for a given n and d.

We say that q is the **quotient** when n is divided by d, and that r is the **remainder** when n is divided by d.

In Python, for given integers `n` and `d`, we can compute their quotient using `//`, their remainder using `%`, and both at the same time using the built-in function `divmod`:

```
>>> 9 // 2
4
>>> 9 % 2
1
>>> divmod(9, 2)
(4, 1)
```

Modular arithmetic

The final definition in this section introduces some notation that is extremely commonplace in number theory, and by extension in many areas of computer science. Often when we are dealing with relationships between numbers, divisibility is too coarse a relationship: as a predicate, it is constrained by the binary nature of its output. Instead, we often care about the *remainder* when we divide a number by another.

Definition. Let $a, b, n \in \mathbb{Z}$ and assume $n \neq 0$. We say that a is **equivalent to b modulo n** when $n \mid a - b$. In this case, we write $a \equiv b \pmod{n}$.[2]

[2] One warning: the notation $a \equiv b \pmod{n}$ is not exactly the same as `mod` or `%` operator you are familiar with from programming; here, both a and b could be much larger than n, or even negative.

There are two related reasons why this notation is so useful in number theory. The first is that modular equivalence can be used to divide up numbers based on their remainders when divided by n:

Theorem. Let $a, b, n \in \mathbb{Z}$ with $n \neq 0$. Then $a \equiv b \pmod{n}$ if and only if a and b have the same remainder when divided by n.[3]

[3] In Python, we could represent this as the expression `a % n == b % n`.

The second reason this is so useful is that almost all of the "standard" intuitions we have about equality transfer over this new notation as well, making it pretty easy to work with right at the very start.

Theorem. Let $a, b, c, n \in \mathbb{Z}$ with $n \neq 0$. Then the following hold:

1. $a \equiv a \pmod{n}$.
2. If $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$.
3. If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then $a \equiv c \pmod{n}$.

Theorem. Let $a, b, c, d, n \in \mathbb{Z}$ with $n \neq 0$. If $a \equiv c \pmod{n}$ and $b \equiv d \pmod{n}$, then the following hold:

1. $a + b \equiv c + d \pmod{n}$.
2. $a - b \equiv c - d \pmod{n}$.
3. $ab \equiv cd \pmod{n}$.

Note that this second theorem shows that the familiar addition, subtraction, and multiplication operations preserve modular equivalence relationships. However, as we'll study further in this chapter, this is *not* the case with division!